ThanCad: a 2dimensional CAD for engineers

Thanasis Stamos

School of Civil Engineering, National Technical University of Athens, Zografou, Athens, Greece

Abstract

ThanCad is 2dimensional CAD aimed to meet the, ever growing, specific needs of civil and surveying engineers. It is largely command compatible with the leading commercial CAD, but it differentiates to a few concepts such as hierarchical layers, and lack of elements attributes, which is the CAD equivalent to structured programming. ThanCad adds some productivity tools such as line continuation, layer selection and cross-save/read undo mechanism. ThanCad harnesses the power of Python to shrink the development time and the volume of code; to implement and test new ideas in virtually no time; to make ThanCad programmable without the need of separate libraries, plug-ins, special languages, or special OSes; to make 32bit/64bit processor, OS and OS version irrelevant. ThanCad uses Tkinter, the defacto GUI/drawing standard for python, in order not to reinvent the wheel and to achieve platform independence. Several concepts were addressed such as compound elements, text in arbitrary directions, cursor/cross hair, zoom, image zoom, coordinate systems tracking, image resolution, draw order, object snap, different element intersection, mouse wheel windows/Linux differences, input from command window and/or GUI, hierarchical modal windows. Finally, ThanCad uses Python's object oriented programming, but sometimes it follows the Zen of Python and the Linux kernel's philosophy, practicality beats purity.

Introduction

Computer Aided Design (CAD) was introduced to the masses about a quarter century ago with the advent of Personal Computer and the development of an initially small but usable CAD system, AutoCAD[T]. The CAD's strength was, and still is, the capability to make easy modifications to the drawings, without destroying or uglying the original. The actual drawing "with the computer" was sometimes more time consuming than drawing by the hand of an experienced draftsman.

Since then, CAD was come a long way, and it is now easier to draw with computer than by hand, most notably with the introduction of application specific CAD, in areas such as topography, photogrammetry, buildings' architecture, statics, highway design, hydraulics, electronic integration etc. Application specific CAD may be developed from scratch, or it may be based on a general purpose CAD, like AutoCAD[T], which has the capability to be programmable or scriptable. Indeed AutoCAD[T] could traditionally be scripted with AutoLisp[T], a dialect of the programming language Lisp. Later, Visual Basic[T] for Applications was added, Python was contributed by the community (Mischler Georg 2003), and APIs for C and other languages were made available.

The scripting capabilities, useful though they are, have the following disadvantages:

- They are either awkward to program, or expensive or incomplete.
- Some of the scripting languages are proprietary.
- They are limited to the WINDOWS[T] OS.
- The general purpose CAD which is used as a base, is often very expensive.
- Some features, as hierarchical layers, can not be scripted; the code of the CAD itself has to be changed.

For these reasons a new general purpose CAD, ThanCad, was developed from scratch. It will provide more Than conventional CAD, since one of its goals is to be the base and the testbed for other, more specialized CAD systems.

Python

Python (Rossum V. Guido et al 2007) was and still is the ideal computer language for ThanCad. The syntax is clean and extremely readable. It is a fully object oriented language. It provides essential data structures such as lists, sets, dictionaries as builtin objects. It has an extensive and useful standard library (e.g. weak references, iterators, logging etc.). It is cross platform with the true meaning of the word. It can be extended by writing code in C or other compiled languages. The volume of code is an order of magnitude less than compiled languages. On the negative side python is slow, but critical code can be written in C. All in all, python offers rapid prototyping of applications and rapid application development. When ThanCad was conceived, the thought was to prototype it with Python and then port it to C++. Several years later, ThanCad is still in pure Python.

Tkinter

Clearly a graphics package is essential for a CAD. Initially wxWindows (Smart Julian et al 2007), later named wxWidgets, seemed the best solution. WxPython (Dunn R.P. et al 2007) is the interface between Python and wsWidgets. WxPython provides a very reach set of widgets and a method to draw lines, arcs etc. to any graphics window. However at that time, wxPython had some bugs with the fonts, it was very hard to install in Linux Operating System, and the Linux version was slower and less robust than the WINDOWS[T] version. Since one of the goals of ThanCad was to (at least) run well in Linux, wxPython was dropped for Tkinter (Rossum V. Guido et al 2007), which is the python interface to the tk graphics toolkit (Ousterhout John et al 2006).

Tkinter is the defacto graphics standard for Python and although it is spartan for a graphics package, it is very robust, extensible and fast (for an interpreted language). It provides all the basic widgets (menus, buttons,etc.) and a “canvas” which the drawing can be displayed on. The canvas has many primitive capabilities which can be used to build the easy (alas programmatically complex) CAD interface which the users take for granted. Although Tkinter is currently the only graphic package used in ThanCad, a distinction is made between graphics dependent and graphics independent code. Thus, it should be easier to port ThanCad to a different graphics package such as wxWidgets, GTK or Qt in the future.

ThanCad Concepts

ThanCad has many of the usual capabilities of a general purpose CAD. It supports the basic elements - lines, circles, arcs, points, text, images. Currently it has limited support for roads, in the civil engineering sense. The elements may be moved, scaled, rotated, copied, broken to pieces,

joined, copied and pasted to clipboard. Elements can be selected with nearest, window, crossing, layer, previous. Object snap is supported – currently endpoint, midpoint, center, node, quadrant, intersection, tangent, nearest. Queries include distance, angle, list, point id and text find. The text supports fonts, arbitrary direction, backwards, upside/down, vertical, width factor, oblique angle. Various zoom and pan capabilities are implemented – window, all, factor, real time. True color is used throughout ThanCad. User input is accepted through the GUI (mouse) or the command line (keyboard). An unlimited do/undo/redo mechanism is partially implemented. The Save/Open (to an internal format) operations are fast. ThanCad can import dxf files and export to dxf files and various types of image files. There can be many opened drawings simultaneously. Finally a few experimental engineering applications are integrated (grid drafting, map rectification and raster line tracing), which they were really standalone external applications that were quickly embedded into ThanCad.

ThanCad has also a few simple, but surprisingly missing from other CADs, conveniences like previous line continuation, line continuation, angle inquiry, unlimited redo, persisting do/undo mechanism, select that survives undo/redo and user defined image dimensions in export image. These were easily implemented, given that the source code of ThanCad was available for inspection and modification.

ThanCad has also some features that require closer attention as described below.

- Nested Layers: Each ThanCad drawing has one or more layers. A layer is an object which contains set of elements elements. Thus the elements may be grouped by layer so that they can be mass handled. Furthermore, the layers may be nested which leads to layer hierarchy, much like the directory structure of a file system. This makes sense in big drawings, where the number of elements and layers is big. If only one level of layers is allowed, like conventional CADs, the number of layers is so big that the user loses control. The layer hierarchy allows for a great number layers, but keeps the number of layers small at each level.

- Layers' attributes: Each layer also has a set of attributes, such a color, visibility, draw order etc, which are stored in a python dictionary as attribute-name attribute-value pair. Thus it is particularly easy to add new attributes should an application need them. For example a screen-color attribute might be defined, so that elements are drawn with this color on screen but printed with a different color to a printer/plotter.

- Inherited attributes: Each child layer inherits the attributes of its parent when it is created. The attributes are marked as “inherited” and change when the user changes the value of the parent's attributes. On the other hand the user may override the inherited attributes and give “personal” values to all or some of the child layer's attributes. Thus when the user changes the value of an attribute, this value is propagated to its children and its children's children attributes while they are marked as inherited. The propagation stops when a “personal” value is encountered. Finally if an attribute is marked “inherited” and then it is marked “personal”, ThanCad remembers the previous personal value and sets it by default.

- Forced attributes: Not all attributes obey the previous rule. For some attributes, which take on/off values, such as the visibility attribute, it makes sense to unconditionally propagate the off value, regardless of personal value or not. Thus a layer's visibility is on or off (according to its personal value) when its parent's visibility is on. However, the layer's visibility is unconditionally set to off when its parent's visibility is off. When the parent's visibility is set on again, the layer's visibility retains its original value.

- Other attributes: A third category of attributes exist. These attributes which may not be modified by the user. Instead, they can be modified by programs embedded in ThanCad.

Currently only one attribute belongs to this category. The “protected” attribute, if on, prevents creation, deletion or modification of elements inside the layer. The intention is, that if an embedded program depends on, for example, the existence of circle with certain radius, to prevent the user to delete it by accident.

- Elements' attributes: The attributes of an element are the attributes of the layer it belongs to. Different attributes for every element are not supported by ThanCad, because it is the analogous of spaghetti programming in a drawing. It is far better to group related elements into a layer and then set the color of this layer once, than setting the color of each element one by one. Should one element be an exception, a new child layer may be created which will have all the attributes automatically inherited except the color.

- Draw order: In ThanCad it is possible to explicitly define the order with which the elements (or rather the layers) are drawn, by setting a draw order number. This is useful with raster image because a raster image may easily hide lines, circles etc. The layers with the smaller numbers are drawn first (they are “back”), and the layers with the bigger numbers are drawn after (they are “front”). It is not necessary to define which object is “front” and which is “back”. Draw order takes cares of it.

- Platform independence: ThanCad executes unchanged in Linux and Windows, since it depends only on Python and Tkinter, which are cross platform with the true meaning of the word. In fact it should run, though not tested, in any flavor of UNIX or any other OS that supports Python and Tkinter. This includes practically all processors, architectures and Oses, and lets them compete for their features, not the amount of software that runs on them.

ThanCad structure

The structure of ThanCad is roughly described in the following:

- Opened drawing: It is actually Python tuple of 3 values: Name, Drawing, Window.

- Drawing object: It represents the abstract drawing, but it does not show it on screen. It contains all the elements of the drawing (lines, arcs, circles etc.), its layers, and provides the mechanism to manipulate them, such as element deletion. However the object is passive; it does not do something by itself, something else must call it to do the job. The Drawing object is largely graphics (or gui) independent, or at least gui dependent in a controlled way, as described below.

- Layer hierarchy: Each Drawing object has one or more layer objects organized in hierarchy. A layer object contains set of elements elements which are saved in a Python set and is responsible for the attributes of the layer and elements it contains (for example to propagate the attributes to all its layer children and its elements).

- Element object: Each ThanCad element is defined by a class. All manipulations and queries done on the element are, of course, implemented as methods in these classes. In addition all gui dependent code or code dependent on any external object (for example dxf export or image export) is also implemented as methods. Such methods receive the external object (e.g. Tkinter object decorated with some attributes) and draw the element on to it. The caller of the method does not know the type of the external object it passes to the method or how to handle it. Thus the dependent code is transparent to the Drawing object.

- Window object: Currently a Tkinter gui window, window object displays the drawing, gets user input and uses the Drawing object to perform the user request. For example if the users scales a circle, it locates the element and calls its scale method. After that, a redraw on the window will reflect the change to the screen. However this is a slow operation. So, ThanCad also scales the drawn object, that is the actual circle in a Tkinter Canvas. Clearly the operation done on the abstract element must be synchronized with the operation on the drawn element.

- Dialogs: Some times the user has to enter complex information to ThanCad, such as color definition, which is best done through a graphics dialog. Currently all the dialogs are done with the Tkinter gui.

- Commands: All the written ThanCad commands (for example BREAK) correspond to a function responsible to carry on the command. This is an intermediate layer between Window object and Drawing object in order to reduce the dependence of the command processing to the gui.

- Various: There are of course many details which are not covered by the above, such as intersection of line segments, logging objects, coordinate transformation, options persistence etc.

Python Implementation

ThanCad naturally makes heavy use of Python's builtin objects and Python's object oriented capabilities. In fact, most of the code is inside definitions of classes. Thus, as in almost all python programs, it is relatively easy to make additions and modifications to the code. For example the layer attributes are stored in a python dictionary, and thus the addition of an attribute is the addition to a python dictionary (and some code of what it does). Another example is the Window object, currently Tkinter Window object, which can be replaced by a different one, like Qt Window object.

Although ThanCad uses object oriented programming, and because Python gives the capability, everything is free to call everything else, and sometimes it does. For example the <MouseWheel> event in Windows^{T} OS, for some unknown reason, can not be intercepted by the Tkinter canvas widget. Instead, its parent TopLevel window receives the event, so the canvas must intercept its parent's event with one of its own methods. Here, practicality beats purity.

One of the most interesting python implementation aspects is the intersection of two arbitrary python elements. If there are n classes of elements then that n^2 intersection functions are needed, since each element class may be intersected with any other element class, itself included. Furthermore double dispatch (Horstman Cay 1995) is needed to make the computation of intersection automatic (object oriented). But, with python, there can be many improvements:

1. A python dictionary is created with:
   key: 2dimensional Python tuple with element class 1 and element class 2
   value: The Python function of the intersection of these 2 element classes
   For example:

```
thanIntPair = {     (ThanArc,     ThanArc)     : thanArcArc,
                    (ThanArc,     ThanCircle) : thanArcCircle,
              }
```

Thus the call of the appropriate function is done with a single statement:

```
thanIntPair[e1.__class__, e2.__class__](e1, e2)
```

2. The intersection operation is commutative, that is the intersection a line and an arc is the same as the intersection of an arc and a line. Thus only $\frac{n^2}{2}$ functions are needed. In order to achieve a method of actually inversing the arguments of a function is needed:

```
class __Inv:
     "Calls a function with inverted arguments."
     def __init__(self, func):
          self.func = func
     def __call__(self, e1, e2):
          return self.func(e2, e1)
thanIntPair[ThanCircle, ThanArc] = __Inv(thanArcCircle)
```

3. Python's polymorphism can be used to reduce the intersection functions. For example ThanCad's image element has a rectangle as boundary. This rectangle behaves (on purpose) like a line element. Thus the functions of line intersection may also be used for the image element ("if it quacks like a duck and ducks like a duck then it is duck enough").
4. Finally if an element does not support intersection like points and texts, a do nothing function completes the dictionary.

Tkinter implementation

The gui interface is the heart of any CAD, as well as ThanCad. With Tkinter it was possible build a modern, alas programmatically complex, interface, because although Tkinter is spartan, it gives the necessary building blocks to make almost anything. According to the author this was, and still is, one of the most interesting fields of ThanCad. In the following several problems and their solutions will be presented.

Croshair cursor

ThanCad uses a croshair as mouse cursor. It is a horizontal and a vertical line that span the window and intersect at the mouse point. Unfortunately Tkinter does not have such a mouse cursor, so it is created as two Tkinter Canvas lines. The length of each line is determined by Canvas winfo_width() and winfo_height() functions which return the window dimensions. These function are called when the window is created and any time the window changes dimensions, via the <configure> event interception. Sometimes these functions return zero. In order to prevent that, it was empirically found that the Canvas update_idletasks() must be preceded. For Windows[T] OS the more dangerous function update() must be called, which may lead to race conditions (Lundh Fredrik 1999). Even so, the first time that the window is created, winfo_*() will not work. In this the Tkinter system must be left some time to stabilize, so the call to the cursor resize() function must be delayed with the Canvas after() function:

```
canvas.after(DT, crosHair.resize)
```

DT may be 200 ms for Python and this millennium' s hardware.

Object real time moving

When the users want to transfer some objects from one position to another, they expect to see the objects actually moving as they move the cursor. This can be done by intercepting the event <motion> which is triggered when the mouse moves. Then the objects are deleted and they are redrawn in the new mouse position, giving the illusion of moving. This can be quite slow if the

Tkinter canvas contains a very big amount of elements. A little faster way is to use the canvas move() function for the chosen elements.

Yet, the fastest way is to actually alter the coordinates of the object with the canvas coords() function. Of course the computation of the new coordinates must be also fast (for example via a list comprehension) and the old coordinates must be known without a second call. Otherwise the gain is neutralized or even negated.

Real time panning

This is really like moving all the objects in the opposite direction. However, Tkinter provides the xview() and yview() functions which pan the canvas in the x or y direction, and do this a little faster than the move() function.

Object real time scaling

When the users want to scale some objects, they expect to see the objects actually growing as they move the cursor. This can be done by intercepting the event <motion> which is triggered when the mouse moves. Then the objects are deleted and they are redrawn with bigger (or smaller) size. This can be quite slow if the Tkinter canvas contains a very big amount of elements. A little faster way is to use the canvas scale() function for the chosen elements.

Yet, the fastest way is to actually alter the coordinates of the element with the canvas coords() function. Of course the computation of the new coordinates must be also fast (for example via a list comprehension) and the old coordinates must be known without a second call. Otherwise the gain is neutralized or even negated.

Zooming

Unfortunately Tkinter does not provide functions to zoom itself. The zoom is done scaling all the objects, bigger if zooming in, or smaller if zooming out. This creates the problem that the actually drawn canvas object has no longer the same coordinates as the abstract ThanCad element. Thus ThanCad has to cope with 2 coordinate systems. Worse, the canvas coordinates are different to the pixel coordinates by a constant dx, dy. Thus ThanCad hat to keep track of 3 coordinate systems.

Dragging

The pan and the zoom is done while the user is dragging the mouse, which means that the user has to move the mouse with the button pressed in order to pan or zoom. Each mouse drag event zooms the drawing by a specified factor and ThanCad used to track these events in order to find the overall zoom factor. But, rarely, some events are lost, and thus ThanCad did not compute the correct overall factor, which meant that the 3 coordinate system were no longer synchronized. The solution to this problem is to create 2 small lines at defined positions, well outside the visible window. These lines, which are not visible, are zoomed with every other object. Then it is possible to compute the real overall scale factor examining the new positions of these small lines. After that the lines are not needed and they are deleted.

Sequential input in environment with threads

ThanCad has a small number of basic functions which get user input from the Tkinter Window, such as getpoint() which waits for the user to press click and returns the coordinates of the cursor, and getline() which does the same job and also displays a line from the previous point to the cursor. These functions may be called by other code to get complex input. For example to get a line segment, first getpoint() is called to get the first point of the segment and then getline() is called to get the last point of the segment.

The problem is that between getpoint() and getline() the program must yield control to the Tkinter graphics mechanism and when the user clicks, the program must execute sequentially the next function, getline(). If getpoint() yields control to Tkinter, i.e. it terminates, how can the sequential flow continue? If getpoint() does not terminate, it will not be possible to even use the

mouse.

The, dirty, solution ThanCad uses, is that the get*() functions continuously call the canvas update() function, while a state variable is not None. The canvas update() function completes any pending Tkinter jobs such as mouse move or mouse click, and then returns. When the user clicks, the <Button-1> event is triggered, ThanCad intercepts it, records the coordinates of the mouse, and sets the state variable to None. The get*() function notices that the state variable is None, gets the recorded coordinates and returns them to the caller.

Text

Unfortunately Tkinter does not allow arbitrary direction of text. In order to work around this problem, ThanCad creates characters as set of lines. The movement, rotation and scaling of the character lines to their correct location are done in python and tends to be slow for a large number of characters. Worse, each line is separate Tkinter object which slows Tkinter operations such as pan. Thus, ThanCad, represents text with height less than 4 pixel, with a rectangle. Zoom in and regeneration of the drawing will make text readable again.

Compound elements

ThanCad text is made of many Tkinter objects and it is called compound. Compound elements are also point (represented by cross, or triangle etc.) and image (which is the image and rectangle as the boundary of the image). The Tkinter objects of a compound element must be treated the same way. Tkinter allows this to be done by assigning the same “attribute”, which is just a text, to all the objects of the compound element. Using this attribute, the objects may be moved, scaled or deleted with a single Tkinter call.

Compound elements need special treatment in ThanCad's select crossing command. This command is implemented calling Tkinter find_overlapping() function, which finds all Tkinter objects in, or partially in, a given rectangle. If a found Tkinter object is part of a compound element, then ThanCad manually adds alls the objects with the same attribute to the selection.

Compound elements also need special treatment in ThanCad's select window command. This command is implemented calling Tkinter find_enclosing() function, which finds all Tkinter objects in a given rectangle. If a found Tkinter object is part of a compound element, then ThanCad checks if the set of all objects with the same attribute is a subset of the selection. If it is not, the set is subtracted from the selection.

Mouse wheel handling

ThanCad zooms the drawing in or out if the user rotates the mouse wheel. In Linux the mouse wheel rotation triggers the <Button-4> or <Button-5> according to rotation spin. In Windows^{T} the <MouseWheel> event is triggered with the variable event.delta set to a value less or greater than zero according to rotation spin. It is not difficult to factor out the common code for Linux and Windows^{T}, it is just a minor inconvenience.

But it seems that the <MouseWheel> event is triggered only by the parent TopLevel window of the Tkinter canvas, and not by the Tkinter canvas itself. Which means that TopLevel's <MouseWheel> must be intercepted by a method of canvas (inelegant). Also, the pixel coordinates of the cursor must be translated from the coordinate system of the TopLevel window to the coordinate system of the canvas. This is done by subtracting the pixel coordinates of the top-left corner of the canvas from the pixel coordinates of the cursor returned by the event.

Object snap

Object snap is the the capability of ThanCad to snap on a specific point of an element (endpoint, midpoint, center etc), when the user places the cursor near it. When object snap is on, and the mouse moves, the <Motion> event is triggered, and ThanCad calls find_overlapping() to locate an element near the cursor. If a near element is found, and the element supports the specific object snap (for example a circle has no endpoints), the specific point is highlighted (with a rectange, cross, circle

etc) and the coordinates of this point are saved. If the user clicks, then the saved coordinates are returned, not the coordinates of the cursor (where the mouse is).

Hierarchical modal windows

A modal window is a window that disables the input capabilities of all the windows except itself. Modal windows are created by calling their grab_set() method. When the modal window dies, the grab_release() function is called and the other windows may accept input.

In ThanCad it is usual to display a modal window which disables the input of the canvas, then a second modal window which disables the input of the first modal window and the canvas and so on. However when the second modal window dies, the input of all windows are automatically enabled, so the first modal window is no loner modal.

ThanCad provides a mechanism that supports hierarchical modal windows, which automatically retain the modal feature. In the above example, when the second modal window dies, the grab_set() function is automatically called for the first one, and thus it becomes modal once again.

As usual, the Python implementation is simple and brief. In fact it so brief that the actual code is given below. A list of weak references to all modal windows is kept (in a module). When the most recent modal window dies, its weak reference is invalidated. The Python weak reference mechanism automatically calls a housekeeping function, which deletes the dead window from the list, and calls grab_set() for the previous modal window found in the list.

The only complication is that any window may be deleted by the window manager (for example pressing the cross button of the window), an action which should only be allowed for the most recent modal window. So the WM_DELETE_WINDOW event is intercepted and it is disabled for all modal windows except for the most recent. When a modal window is activated again the WM_DELETE_WINDOW event is re-enabled. Here it is assumed that the modal windows will have a cancel() method which be called when the user deletes the window via the window manager.

```
__grabWins = []
def thanGrabSet(win):
    "Perform a nested grab_set."
    if len(__grabWins) > 0:
        win1 = __grabWins[-1]
        win1().grab_release()
        win1().protocol("WM_DELETE_WINDOW", lambda: "break")
    win.update()
    win.grab_set()
    win1 = weakref.ref(win, __grabWinDied)
    __grabWins.append(win1)

def __grabWinDied(weakwin):
    "This is called when the window which has the grab dies."
    assert len(__grabWins) > 0 and __grabWins[-1] == weakwin, "How
did this happen?"
    del __grabWins[-1]
    if len(__grabWins) <= 0: return
    win1 = __grabWins[-1]
    win1().protocol("WM_DELETE_WINDOW", win1().cancel)
```

```
win1().lift()
win1().focus_set()
win1().grab_set()
```

Ideas for the future

Fuse file system
The layer hierarchy closely resembles a file system, where the layers are directories and elements are files. Indeed it is possible to create such a virtual file system with help of FUSE – File system in USEr space (Szeredi Miklos et al 2007). FUSE has python bindings so it should be relatively easy to implement it. The contents of each file will be the geometric properties of the element it represents. For example a circle will have the coordinates of its center and its radius, plus some comments to make the file readable. Each directory, which will correspond to a layer, will have a special file which will contain the attributes the layer. The virtual file system will make it trivial for any external application to create new drawings or modify existing. This may even be done in real time by utilizing the Linux kernel's inotify subsystem (Martini Sebastien et al 2007). FUSE is available for Linux, FreeBsd, OpenSolaris, MACOS-X. Alternatively libfrerris (Martin Ben et al 2007) may be used for the same result.

Client server approach
It is usual for big civil engineer drawings to demand several hours for editing. In order to save time, the drawing is given to more than one user, who make changes to separate parts of the drawings. After that all the changes are merged by hand. This many times is awkward and error prone. If each user holds backup history of the drawing, then the whole operation is a maintenance nightmare. What is needed, is a way for many users to edit the same drawing.

ThanCad code may be split to a server which holds the elements of a drawing and a client which displays the drawing and takes input from the user. The idea is to have one common server and many clients to the same drawing. When a user modifies a set of elements, the elements will be locked, preventing other users from accessing them. When the first user completes the changes, they are immediately displayed on all clients. The save operation will be handled by the server. However the do/undo mechanism will be much more complicated.

Other GUIs - Web
It should not be prohibitably difficult to add another GUI like Qt in ThanCad, to see if the result is better or faster. In particular, it might be a good idea to use Firefox as a GUI and see if ThanCad can be transformed to a web application, like Google^{T} Docs and Spreadsheets (Google 2007), and see if whether this is beneficial or not.

Embedded application support
Many engineering applications handle essentially CAD elements without a CAD. An example is the computation of the coordinate transformation parameters between two sets of coordinates, which both correspond to the same points on earth, but each set corresponds to a different coordinate systems. The computation is strictly mathematical and does not relate to CAD at all, but the error messages could be better understood, if the points with errors were graphically shown and were highlighted with an attached error explanation. These little conveniences should be collected into an API, to ease existing applications' embedding.

Typical use improvement
In order to improve the aspects of ThanCad most users use, at first these aspects must be found out.

So it might be possible to record, after asking politely the users for permission, typical user behavior (users click a lot or type a lot, which capabilities they use most, which elements, how many times do they save, what OS they use, where is most of the user time spent, where is the most processor time spent, and so on) and typical drawing complexity (what kind of elements it has, how many elements, how much memory it consumes, etc). This information will be used to focus development to the features the users want, to improve algorithms of frequently used operations, to add conveniences and generally improve the user experience with ThanCad. This information will also be of immense value to the academia.

Trademarks

References

Dunn R.P. et al 2007, wxPython a Python extension module that wraps wxWidgets, http://www.wxpython.org/, Last Update: 14/05/2007

Google 2007, Google web applications, http://www.google.com/a/, Last Update 2007

Martin Ben et al 2007, Libferris is a virtual file system (VFS) that runs in user space, http://witme.sourceforge.net/libferris.web/, Last Update 13/06/2007

Martini Sebastien et al 2007, Pyinotify is a Python module for watching filesystems changes, http://pyinotify.sourceforge.net/, Last Update 17/02/2007

Rossum V. Guido et al 2007, Python Programming Language, http://www.python.org/, Last Update: 2/07/2007

Smart Julian et al 2007, wxWidgets project, http://www.wxwidgets.org/, Last Update: 4/04/2007

Szeredi Miklos et al 2007, Filesystem in Userspace (FUSE), http://fuse.sourceforge.net, Last Update: 5/02/2007

Ousterhout John et al 2006, Tk an open source cross-platform widget, http://www.tcl.tk, Last Update: 19/10/2006

Mischler Georg 2003, Autocad/Intellicad Extension for Python, http://pyacad.sourceforge.net/, Last Update: 10/07/2003

Lundh Fredrik 1999, An introduction to Tkinter, electronic book, http://www.pythonware.com/library/tkinter/introduction/, Last Update 1999

Horstman Cay 1995, Mastering Object-Oriented Design in C++, John Wiley & Sons Inc., ISBN 0-471-59484-9